P0190R4: Proposal for New

memory_order_consume Definition

Doc. No.:          WG21/P0190R4
Date:             2017-02-07
Reply to:        Paul E. McKenney, Michael Wong, Hans Boehm,
                  Jens Maurer, Jeffrey Yasskin, and JF Bastien
Email:           paulmck@linux.vnet.ibm.com, michael@codeplay.com,
                  boehm@acm.org, Jens.Maurer@gmx.net,
                  jyasskin@google.com, and jfbastien@apple.com

Other contributors:
Torvald Riegel, Jeff Preshing, Clark Nelson, Olivier Giroux,
Lawrence Crowl, Alec Teal, David Howells, David Lang,
George Spelvin, Jeff Law, Joseph S. Myers, Linus Torvalds,
Mark Batty, Michael Matz, Peter Sewell, Peter Zijlstra,
Ramana Radhakrishnan, Richard Biener, Will Deacon, Faisal Vali,
Behan Webster, Tony Tye, Thomas Kopepe, Boqun Feng, ...

July 28, 2017

This document is a update to WG21/P0190R3, which is an update to
WG21/P0190R2, which is an update to WG21/P0190R1, in all cases based on
e-mail discussions. WG21/P0190R1 is itself an update to WG21/P0190R0, based
on discussions at the Jacksonville meeting. That document was in turn a follow-on
to WG21/P0098R1, based on email discussion and on discussions at the 2015
meeting at Kona, which should be consulted for background on memory_order_consume and for a number of alternatives to the definition in the standard [25].
The main purpose of memory_order_consume is to provide language support for
read-copy update (RCU) [5, 10, 12, 13, 14, 16, 18], which is heavily used in the
Linux kernel and which is seeing increasing use in user-level multi-threaded soft-
ware [1, 2, 4, 7, 8, 21, 24]. In addition, the intersection of RCU and transactional
memory is starting to garner significant attention [9, 11, 19, 20, 22].

However, this proposal has use cases beyond just RCU. For example, Java
final-field accesses provide a closely related dependency guarantee, and a similar
capability will likely be needed by any C++ garbage collector. Furthermore,
garbage collectors enable use cases that are quite similar to those of RCU [6]. In
addition, it is highly likely that additional concurrent code will rely on depen-
dency ordering, given that mainstream implementations provide dependency-ordering guarantees via TSO or via hardware dependency ordering.

Please note that this document focuses only on the semantics of dependency ordering. Information on the syntax of marked dependency chains may be found in the working paper entitled “P0462R1: Marking memory_order_consume Dependency Chains” which updates P0190R2 at http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0462r0.pdf.

This document presents a proposed solution to the problems with memory_order_consume in the current C and C++ standards. Section 1 describes the overall approach, Section 2 provides an informal definition, Section 3 provides draft wording, Section 4 provides a series of litmus tests demonstrating dependency chains, and finally Section 5 summarizes benefits, drawbacks, mitigations, and open issues.

1 Approach

The purpose of memory_order_consume loads is to provide ordering guarantees similar to those of memory_order_acquire loads, but, in contrast to the wording in the current standard, only in cases where there is a robust dependency between the memory_order_consume load and some subsequent operation. A robust dependency is guaranteed to be preserved by both compilers and CPUs (other than DEC Alpha) without the need for explicit memory-fence instructions, and is a subset of the syntactic dependencies that earlier C++ standards [25] specify for memory_order_consume. In contrast, the approach currently in the standard requires the compiler to insert artificial dependencies or even memory-barrier instructions so that even non-robust dependencies will be preserved.\(^1\)

Another conspicuous change is that this proposal does not require dependencies to be carried by integers, but instead only by pointers. Note that versions v4.2 and later of the Linux kernel avoid carrying dependencies through integer variables [15]. At first glance, it might seem that restricted carrying of dependencies through intptr_t and uintptr_t is needed to tag pointers, but such tagging is not supported by the current C++ standard. This document therefore completely prohibits carrying dependencies through integers.

In fact, the overall approach is to provide only those dependency-chain preservation guarantees that are actually used in recent releases of the Linux kernel. This has the beneficial effect of making a minimal implementation (excluding pointer-comparison intrinsics and diagnostics) trivial on systems other than DEC Alpha: The implementation need only compile a memory_order_consume to emit the same code that it would for a memory_order_relaxed load.\(^2\)

\(^1\) Note to Linux kernel hackers: From here on out, the word “implementation” will be used instead of “compiler”, as is the custom in the C++ Standards Committee.

\(^2\) DEC Alpha systems require that each memory_order_consume loads be followed by full memory-barrier instructions if there are any loads that depend on the memory_order_consume load. [3, 23]. Therefore, on DEC Alpha we recommend promoting memory_order_consume loads to memory_order_acquire.
2 Informal Dependency-Chain Definition

This section provides an informal description of the proposed solution. Section 2.1 lists requirements and desiderata, and Section 2.2 provides the informal description, examples, and discussion.

2.1 Requirements and Desiderata

There are a large number of constraints imposed on the problem of defining `memory_order_consume` dependency chains. Many of them stem from the desire to accommodate existing dependency-chain practice, for example within the Linux kernel, but at the same time placing little or (preferably) no burden on implementations. Although invention does have its place, it should be noted that invention was exactly what resulted in an unworkable definition of `memory_order_consume` in the C++11 standard. Requirements stemming from Linux-kernel compatibility include:

1. Dependency chains must not depend solely on marking objects carrying dependencies. Objects include local variables, function parameters, and function return declarations.

2. Although it is perfectly acceptable for dependency chains to mark the load heading the chain, dependency chains must not depend on marking accesses or operations further down the chain.

3. Use of `memory_order_consume` loads should not result in unsolicited memory-fence instructions. High-quality implementations would avoid emitting such instructions, and not-so-high-quality implementations would at the very least be able to issue a warning when such an instruction was emitted.

4. Dependency chains need only be carried through pointer values, however, they must be carried to (not through) non-pointer values in a number of cases.

The Linux kernel requires that dependencies be carried through bit manipulations of pointer values, however, such bit manipulation invokes undefined behavior. Bit manipulation of pointer values is useful for tagging and for some types of memory allocators, and therefore might be worth standardizing in its own right.

Compiler writers require that dependency chains not require explicit tracing by implementations, such tracing being one of the major rocks on which the C++11 definition of `memory_order_consume` foundered.

Although Linux-kernel compatibility requires that dependency chains not depend solely on markings, it is entirely acceptable for such marking to provide additional benefits such as higher-quality diagnostics and software-engineering benefits. In fact, the Linux kernel already provides the `rcu` marking that may optionally be applied to RCU-protected pointers, that is, pointers that are
loaded using \texttt{rcu\_dereference()}, which is the Linux kernel’s counterpart to a \texttt{memory\_order\_consume} load. One key hoped-for benefit of optional markings is ease of formal verification.

2.2 Informal Definition

This informal description covers operations that extend dependency chains (Section 2.2.1) and operations that terminate dependency chains (Section 2.2.2).

2.2.1 Extending Dependency Chains

The following categories of primitive operations extend dependency chains:\(^3\)

1. Moving, copying, and casting.
2. Pointer offsets.
3. Dereferencing and address-of, including class-member access.
4. Miscellaneous operators.

Any other operation terminates a dependency chain. The operations that extend dependency chains are covered in more detail below.

Moving, Copying, and Casting: Values that are part of a dependency chain may be moved, copied, and casted (in some cases), and the dependency chain will propagate to the result.

1. If any pointer value is part of a dependency chain, then using that value as the left-hand side of an assignment expression extends the chain to cover the assignment. This rule is exercised in the Linux kernel by stores into fields making up an RCU protected data element. This is illustrated by Figure 1.

2. If any pointer value is part of a dependency chain, then using that value as the right-hand side of an assignment expression extends the chain to cover both the assignment and the value returned by that assignment statement. Line 20 of Figure 2 shows how this rule may be used to extend a dependency chain into a local variable.

\(^3\) In case of operator overloading, the actual functions called must be analyzed in order to determine their effects on dependency chains.
3. If any pointer value that is part of a dependency chain is stored to a non-shared variable, then any value loaded by a later load from that same variable by that same thread is also part of the dependency chain. Lines 20 and 24 of Figure 2 illustrate this rule, though this rule would apply even if local variable lsp was an `intptr_t` instead of a pointer. Note that the job of determining whether or not a given variable is non-shared falls to the developer, not the implementation. That said, a high-QoI implementation might choose to make this determination in order to issue helpful diagnostic messages.

4. If a pointer value that is part of a dependency chain is stored to any variable, then any pointer value loaded by a later load from that same variable by that same thread is also part of the dependency chain. Lines 20 and 24 of Figure 2 illustrate this rule, though this rule would apply even if local variable lsp was instead a shared variable. As before, determining whether or not the store and load were carried out by the same thread falls to the developer, not to the implementation.

5. If a pointer value is part of a dependency chain, then casting it (either explicitly or implicitly) to any pointer type extends the chain to the result. Such casts are used heavily in the Linux kernel, for example, in the `list_for_each_entry_rcu()` and `list_entry_rcu()` RCU-protected list-traversal C-preprocessor macros.

6. If a pointer value is part of a dependency chain, then if that pointer is used as a pointer-type argument of a function call, the dependency chain extends to the corresponding parameter.

7. If a function returns a pointer value that is part of a dependency chain, the dependency chain extends to the returned value in the calling function.

**Pointer Offsets:** If a given pointer is part of a dependency chain, then integral offsets to that pointer are also part of that dependency chain.

1. If a pointer is part of a dependency chain, then adding an integral value to that pointer extends the chain to the resulting value. This applies for both positive and negative integers, and also to addition via the infix `+` operator and via the postfix `[]` operator. Note that the addition must be carried out on a pointer: Casting to an integral type and then carrying out the addition is permitted to break the dependency chain. Therefore, instead of casting to an integral type to carry out the addition, cast to a pointer to `char`.

4 Line 24 of Figure 2 illustrates this, given that the `->t` acts as a pointer offset prior to indirection.

---

4 Yes, some old systems had strange formats for character pointers, and this restriction does exclude those systems from this nuance of dependency ordering. However, to the best of my knowledge, all such systems were uniprocessors, so this is not a real problem.
1 #define rcu_dereference(x) \
2 atomic_load_explicit((x), memory_order_consume);
3
4 struct liststackhead {
5 struct liststack *first;
6 }
7
8 struct liststack {
9 struct liststack *next;
10 void *t;
11 struct rcu_head rh;
12 }
13
14 void *ls_front(struct liststackhead *head)
15 {
16 void *data;
17 struct liststack *lsp;
18
19 rcu_read_lock();
20 lsp = rcu_dereference(head->first);
21 if (lsp == NULL)
22 data = NULL;
23 else
24 data = rcu_dereference(lsp->t);
25 rcu_read_unlock();
26 return data;
27 }

Figure 2: List-Based-Stack Whole-Program Approach, 1 of 2

2. If a pointer is part of a dependency chain, then subtracting an integer
from that pointer extends the chain to the resulting value. This applies
for both positive and negative integers. Again, casting to an integral type
and then carrying out the subtraction will break the dependency chain,
so instead cast to a pointer to char. The Linux-kernel container_of() macro illustrates this. This macro is used to find the beginning of a
structure given a pointer to a field within that same structure.

3. Note that class-member access operators (. and ->) can be thought of as
computing an offset as part of their execution.

Dereferencing and Address-Of: Dereferencing a pointer that is part of a
dependency chain extends the dependency chain to the result, but only when
the resulting value is a pointer type. Taking the address of a pointer that is part
of a dependency chain, and then dereferencing the resulting pointer, extends the
dependency chain to the result.

1. If a pointer is part of a dependency chain, then dereferencing it using the
prefix * operator extends the chain through the dereference operation.

2. If a pointer is part of a dependency chain, then dereferencing it using the
-> field-selection operator extends the chain to the field. Note that
the when the -> operator is followed by one or more . operators, these
latter operators are equivalent to adding a constant integer to the original
pointer. Line 24 of Figure 2 directly illustrates this rule.
int ls_push(struct liststackhead *head, void *t)
{
    struct liststack *lsp;
    struct liststack *lsnp1;
    struct liststack *lsnp2;
    size_t sz;
    sz = sizeof(*lsp);
    sz = (sz + CACHE_LINE_SIZE - 1) / CACHE_LINE_SIZE;
    sz *= CACHE_LINE_SIZE;
    lsp = (struct liststack *)malloc(sz);
    if (!lsp)
        return -ENOMEM;
    if (!t)
        abort();
    lsp->t = t;
    rcu_read_lock();
    lsnp2 = ACCESS_ONCE(head->first);
    do {
        lsnp1 = lsnp2;
        lsp->next = lsnp1;
        lsnp2 = cmpxchg(&head->first, lsnp1, lsp);
    } while (lsnp1 != lsnp2);
    rcu_read_unlock();
    return 0;
}

static void ls_rcu_free_cb(struct rcu_head *rhp)
{
    struct liststack *lsp;
    lsp = container_of(rhp, struct liststack, rh);
    free(lsp);
}

void *ls_pop(struct liststackhead *head)
{
    struct liststack *lsp;
    struct liststack *lsnp1;
    struct liststack *lsnp2;
    void *data;
    rcu_read_lock();
    lsnp2 = rcu_dereference(head->first);
    do {
        lsnp1 = lsnp2;
        if (lsnp1 == NULL) {
            rcu_read_unlock();
            return NULL;
        }
        lsp = rcu_dereference(lsnp1->next);
        lsnp2 = cmpxchg(&head->first, lsnp1, lsp);
    } while (lsnp1 != lsnp2);
    data = rcu_dereference(lsnp2->t);
    call_rcu(&lsnp2->rh, ls_rcu_free_cb);
    return data;
}

Figure 3: List-Based-Stack Whole-Program Approach, 2 of 2
Figure 4: Back-Propagation of Dependency-Chain Breakage Due to Comparisons

3. If a pointer is part of a dependency chain, then applying the unary &
   address-of operator, optionally casting this address to a pointer type (per-
   haps repeatedly to different pointer types, either explicitly or implicitly),
   then applying the * dereference operator extends the chain to the result.
   This is used by some of the Linux-kernel list-processing macros.

Miscellaneous Operations: The following operations also extend depen-
   dency chains.

1. If a pointer is part of a dependency chain, and that pointer appears in
   the operand of a ?: operator selected by the condition, then the chain
   extends to the result. Please note that ?: does not extend chains from its
   condition, only from its second or third argument.

2. If a pointer is part of a dependency chain, and that pointer appears in the
   right-hand operand of a , operator, then the chain extends to the result.
   Please note that the , operator does not extend chains from its left-hand
   operand, only from its right-hand operand.

3. If a given operation extends a dependency chain, then so does its atomic
   counterpart. For example, the rules applying to assignments also apply
   to atomic loads and stores. It also applies to atomic exchange and atomic
   compare and swap.

2.2.2 Terminating Dependency Chains

Even though all other operations terminate dependency chains, there are a few
that deserve special mention:

1. Equality comparisons.

2. Narrowing magnitude comparisons.

3. Narrowing arithmetic operations.
if (p > &foo)
   do_something(p);
else if (p < &foo)
   do_something_else(p);
else
   do_something_nodep(p);

Figure 5: Inequality-Comparison Dependency-Chain Breakage

4. Narrowing bitwise operations.

5. Passing values between threads without using a memory_order_consume
   load.

6. Undefined behavior.

7. Use of std::kill_dependency.

Each of these is covered below.

Equality comparisons: If a pointer is part of a dependency chain, then a ==
or != comparison that compares equal to some other pointer, where that other
pointer is not part of any dependency chain, will cause any uses of the original
pointer to no longer be part of the dependency chain. This dependency-chain
breakage can back-propagate to earlier uses of the pointer, so that in Figure 4, if
the comparison on line 10 compares equal, then the access on line 9 is not part of
the dependency chain. This is admittedly a rather strange code fragment, and
besides, the Linux-kernel barrier() macro could prevent this if placed between
lines 9 and 10. Furthermore, the Linux kernel’s list macros avoid this situation
because the equal comparison terminates the loop.

So what if the implementation introduces an equality comparison? This
might happen when doing feedback-directed optimization, where the imple-
m entation might notice (for example) that a particularly statically allocated
structure was almost always the first element on a given list. The implemen-
tation might therefore introduce a specialization optimization, comparing the
addresses and generating code using the statically allocated structure on equals
comparison. On the one hand, in the cases where the Linux kernel adds a
statically allocated structure to an RCU-protected linked data structure, that
structure has been initialized at compile time, so that dependency ordering is
not required. On the other hand, this appears to be an extremely dubious opti-
mization for linked data structures: In a great many cases, the added overhead
of the comparison would overwhelm the benefits of generating code based on
the statically allocated structure.

High-quality implementations would therefore be expected to provide means
for disabling this sort of optimization, especially for pointers obtained from
the heap. After all, use of statically allocated structures in RCU-protected
lists could be quite useful during out-of-memory conditions, in which case the
specialization optimization would almost always reduce performance, which is not what optimizations are supposed to be doing.

It is tempting to insist that implementations preserve dependency chains even in the face of equality comparisons. However, such insistence eliminates the possibility of a solution that does not require heavy use of explicit marking and std::kill_dependency(). To see this, recall that if a member of a dependency chain is stored into any variable (be it on the stack, on the heap, wherever), and if that same value is reloaded by that same thread, the dependency chain must be preserved. Allowing successful equality comparisons to break dependency chains is therefore essential to an unmarked solution to the dependency-ordering problem.

Alternatively, an intrinsic could provide comparison, but avoid breaking dependency chains. For example, a bool std::pointer_cmp_eq(T *pd, T* p) intrinsic could compare the two pointers, but preserve dependencies carried by pd even if they compare equal. For completeness, a bool std::pointer_cmp_ne(T *pd, T* p) intrinsic could also be provided for not-equal comparisons. Note that there is no way to compare two pointers carrying dependencies and preserve the dependency for both. This is because there are currently no use cases requiring this, and requiring it would require the implementation to be less efficient in its register usage.

**Narrowing magnitude comparisons:** A series of $>$, $<$, $\geq$, or $\leq$ operators that informs the implementation of the exact value of a pointer causes that pointer to no longer be part of the dependency chain. See Figure 5 for an example of this. On line 6 of this figure, the implementation knows that the value of p is equal to &foo, so although there is dependency ordering to lines 2 and 4, there is no dependency ordering to line 6. This dependency-chain breakage can back-propagate, just as for equality comparisons. However, dependencies are maintained for normal uses, for example, the use of comparisons for deadlock avoidance when acquiring locks contained in multiple RCU-protected data elements.

As with equality comparison, an intrinsic could provide comparison, but avoid breaking dependency chains. For example, bool std::pointer_cmp_gt(T *pd, T* p), bool std::pointer_cmp_ge(T *pd, T* p), bool std::pointer_cmp_lt(T *pd, T* p), and bool std::pointer_cmp_le(T *pd, T* p) intrinsics could compare the two pointers, but preserve dependencies carried by pd even if a series of comparisons allowed the implementation to deduce the exact value of the pointer.

**Narrowing arithmetic operations:** If a pointer is part of a dependency chain, and if the values added to or subtracted from that pointer cancel the pointer value so as to allow the implementation to precisely determine the resulting value, then the resulting value will not be part of any dependency chain. For example, if p is part of a dependency chain, then ((char *)p-(uintptr_t)
1 int *y;
2 int x[1];
3 int foo(void)
4 {
5   int *p;
6   int t;
7   p = atomic_load_explicit(&y, memory_order_consume);
8   t = p - x;
9   return *p;
10 }

Figure 6: Undefined Behavior Breaks Dependency Chain

t) + 65536 will not be.\textsuperscript{5}

**Passing values between threads:** If a value that is part of a dependency chain is stored into a variable by one thread, and loaded from that same variable by some other thread using either a non-atomic load or a \texttt{memory\_order\_relaxed} load, then the dependency chain does not extend to the second thread. To get this effect, the second thread would instead need to use a \texttt{memory\_order\_consume} load. Note that in theory, this would extend the dependency chain even if the corresponding store was a \texttt{memory\_order\_relaxed} store because the required store-side ordering is provided by the dependency chain, however, in practice the C++ standard does not guarantee ordering unless the store also has release semantics. Alternatively, the store might be preceded by an invocation of \texttt{atomic\_thread\_fence()} that provides release semantics.

**Undefined behavior:** If undefined behavior is invoked, then, consistent with the notion of undefined behavior, there are no dependency-chain guarantees.

If a given pointer takes on only one value avoids undefined behavior, then the dependency chain is broken in the same way as it would be in the case of an equality comparison with that same value. This can result in counter-intuitive dependency-chain breakage, as shown in Figure 6. Here, the pointer subtraction on line 10 results in undefined behavior unless \texttt{y} references some element of \texttt{x}. But since there is only one element of \texttt{x}, undefined behavior is avoided only if \texttt{y == &x[0]}, which means that the implementation knows the exact value of \texttt{y} for any valid execution. The implementation is therefore within its rights to replace line 11 with \texttt{return x[0]}, thus breaking the dependency chain.

The moral of this story is “Don’t let singleton arrays anywhere near a value that carries a dependency.”

\texttt{kill\_dependency()}: The result of calling \texttt{std::kill\_dependency} is never part of any dependency chain. This operation can be used to suppress di-

\textsuperscript{5} That said, 5.7p4 of C++ and 6.5.6p8 of C both say that indexing outside of an object is undefined behavior, so the loss of dependency ordering is likely the least of the problems here.
agnostics that implementations might omit for likely misuses of dependency ordering.

3 Draft Wording for Restricted Dependency Chains

This proposal recommends changes to 1.10p11 (Section 3.1), changes to 1.10p12 (Section 3.2), and, optionally, a new 29.9 (Section 3.3).

3.1 Wording for 1.10p11

1.10p11 of the most recent draft of the standard [25] is replaced with the following:

A carries a dependency to B, where A and B are evaluations, side effects, memory loads, or memory stores, and A is of pointer type if:

1. A carries a dependency to X, and X carries a dependency to B, then A carries a dependency to B.
2. An evaluation of an expression A carries a dependency to an evaluation of an expression B, where B is either a prvalue of pointer type or a glvalue, if
   - B is a conditional expression (5.16 [expr.cond]) and A is the second or third operand selected by the first expression or
   - B is a comma expression (5.19 [expr.comma]) and A is its right operand.
3. An evaluation of a glvalue expression A carries a dependency to an evaluation B if
   - B is a class-member access (5.2.5 [expr.ref]) and A is its object expression, or
   - B is the result of an lvalue-to-rvalue conversion (4.1 [conv.lval]) and A is its glvalue operand,
   - B is a unary & expression (5.3.1 [expr.unary.op]) and A is its operand, or
   - B is a assignment expression (5.18 [expr.ass]) and A is its left operand. [Note: This also covers (p->a = 5) = 7 such that if p is in a dependency chain, then both assignments will also be in that dependency chain. – end note]
4. An evaluation of an expression A carries a dependency to an evaluation B and to the side effect of B (if any) if
   - B is a new-expression (5.3.4 [expr.new]) and A is the invocation of the allocation function, or
• B is an increment or decrement expression (5.2.6 [expr.post.incr] and 5.3.2 [expr.pre.incr]) and A is its operand, or
• B is an additive expression (5.7 [expr.add]), A is one of its operands, and the other operand is of integral type, or
• B is an indirection expression (5.3.1 [expr.unary.op]) and A is its operand, or
• B is a standard conversion (Clause 4), a static cast (5.2.9 [expr.static.cast]), or a const cast (5.2.11 [expr.const.cast]) and A is the operand for the conversion [ Note: An explicit type conversion (5.2.3 [expr.type.conv], 5.4 [expr.cast]) is analyzed in its decomposed form. A reinterpret_cast (5.2.7 [expr.dynamic.cast]) or a dynamic_cast (5.2.10 [expr.reinterpret.cast]) break the dependency chain. – end note ], or
• B is an initialization and A is its initializer, or
• B is a simple assignment expression (5.18 [expr.ass]) and A is its right operand.

5. An evaluation or side effect A carries a dependency to an evaluation of an expression B of pointer type if A is sequenced before B, B is an lvalue-to-rvalue conversion (4.1 [conv.lval]), and A is either
• the side effect that stored the value read by B or
• the evaluation that computed the value read by B.

6. In the following list, a reference to a member function of an atomic type T also applies to the corresponding non-member function; see 29.6.5 [atomics.types.operations.req].
   (a) Given an atomic type T, an evaluation A carries a dependency to a side-effect or load or store C if
   • C is the load resulting from invoking a T::load function and A is the argument for the function’s object parameter, or
   • C is the store resulting from invoking a T::store function and A is the argument for the function’s object parameter, or
   • C is the load or store resulting from a invoking an atomic read-modify-write operation and A is the argument for the function’s object parameter.
   [ Note: this covers T::exchange, the compare-and-exchange functions, and the T::fetch_key functions – end note ]
   (b) If T is an atomic pointer type, an evaluation A also carries a dependency to a side effect or load or store C in the list above if A is the argument for the function’s desired, expected, or operand parameters.
(c) Given an atomic pointer type T, a side effect A carries a
dependency to an evaluation B if A is the side effect that
stored the value read by B and A is sequenced before B and
• B is an invocation of a T::load function, or
• B is an invocation of an atomic read-modify-write op-
eration that returns the value stored by A.

[ Note: Overloaded operators are treated as function calls (13.5
[over.oper]). – end note ]

[ Note: Implicit conversions and implicit invocations of functions are
considered separate evaluations for the purposes of this definition. –
end note ]

[ Note: “Carries a dependency to” is a subset of “is sequenced be-
fore”, and is similarly strictly intra-thread. – end note ]

[ Note: The above rules imply that dependency chains are carries
into and out of functions parameters and return values of pointer
type.

```cpp
int * gp;
int * f(int * p) { return p; }
int g() {
    int * p = atomic_load_explicit(gp, memory_order_consume);
    *f(p) = 5;
}
```

In the preceding example, a dependency would be carried both into
the call to f() through its parameter p and back out through its re-
turn value, so that a dependency would be carried to the assignment
of the value 5. – end note ]

If the implementation is able to exactly determine the value of a
pointer, for example, due to undefined-behavior analysis or due to a
series of comparison operators that enables exact determination of
its value, then no dependency is carried to that pointer.

[ Note: To carry out comparisons without destroying the dependency
chain, use the std::pointer_cmp_eq_dep(), std::pointer_cmp_ne_-
dep(), std::pointer_cmp_gt_dep(), std::pointer_cmp_ge_de-
ep(), std::pointer_cmp_lt_dep(), or std::pointer_cmp_le_de-
intrinsics if they are provided. – end note ]

If A is the argument to std::kill_dependency() and B its result,
no dependency is carried from A to B.

[ Note: The intent of these rules is to enable implementations to
carry out their normal optimizations, while still permitting devel-
opers to rely on the common dependency-ordering use cases. – end
note ]
[Note: Although the [[carries_dependency]] attribute⁶ is no longer needed to specify that a dependency chain exists, this attribute can help the implementation provide higher-quality diagnostics. For example, doing an equality comparison to a non-NULL pointer that is dereferenced could result in a warning diagnostic. Such a warning would make the user aware that no dependency was carried to that pointer. – end note]

Note that some adjustments will be needed for C:

1. Union-based type punning is allowed in C, so will need to be addressed.

There has been some discussion of a marking scheme that avoids loss of dependency when the compiler exactly determines the value of an object. There are some differences of opinion as to whether it is really needed, and, if it is, exactly what form the corresponding wording would take.

### 3.2 Wording for 1.10p12

1.10p12 of the most recent draft is updated by inserting the phrase “of pointer type”, resulting in the following:

An evaluation A is dependency-ordered before an evaluation B if

- A performs a release operation on an atomic object M of pointer type, and, in another thread, B performs a consume operation on M and reads a value written by any side effect in the release sequence headed by A, or
- for some evaluation X, A is dependency-ordered before X and X carries a dependency to B.

### 3.3 Optional Wording for 29.9

A new section 29.9 could add intrinsics for dependency-preserving comparisons:

```
extern "C" bool pointer_cmp_eq_dep(T *pd, T *p) noexcept;
extern "C" bool pointer_cmp_ne_dep(T *pd, T *p) noexcept;
extern "C" bool pointer_cmp_gep(T *pd, T *p) noexcept;
extern "C" bool pointer_cmp_gep(T *pd, T *p) noexcept;
extern "C" bool pointer_cmp_gt(T *pd, T *p) noexcept;
extern "C" bool pointer_cmp_le(T *pd, T *p) noexcept;
```

**Effects:** carries out the specified comparison of the two pointers while preserving any dependencies carried through pd. Note that dependencies carried through p may be lost.

An alternative approach would be to rely on variable marking to get the same effect, thus dispensing with the above intrinsics.

⁶ Some spelling other than [[carries_dependency]] might well be chosen at some point.
4 Litmus Tests

Figure 7 shows some common definitions used by multiple litmus tests, each of which is discussed in one of the following sections.

4.1 Simple Left-Hand-Side Dependency

Figure 8 shows a simple left-hand-side dependency. Dependency ordering guarantees that the assertion on line 8 never triggers.

4.2 Simple Right-Hand-Side Dependency

Figure 9 shows a simple right-hand-side dependency. Dependency ordering guarantees that the assertion on line 17 never triggers.

4.3 Local Storage For Dependency

Figure 9 also demonstrates that storing a pointer to a local variable preserves the dependency chain. The pointer stored to local variable p on line 15 is part of a dependency chain, and the dependency chain remains when it is reloaded on line 17. Dependency ordering thus continues to guarantee that the assertion on line 8 never triggers.

4.4 Non-Local Storage For Dependency

Figure 10 shows that storing to a non-local variable preserves a dependency if this value is later reloaded by the same thread. Dependency ordering guarantees
void thread0(void)
{
    struct rcutest *p;
    p = (struct rcutest *)malloc(sizeof(*p));
    assert(p);
    p->a = 42;
    assert(p->a != 43);
    rcu_assign_pointer(gp, p);
}

void thread1(void)
{
    struct rcutest *p;
    p = rcu_dereference(gp);
    if (p)
    assert(p->a == 42);
    p->a = 43;
}

Figure 8: Litmus Test: Simple Left-Hand-Side Dependency

Figure 9: Litmus Test: Simple Right-Hand-Side Dependency
void thread0(void) {
    struct rcutest *p;
    p = (struct rcutest *)malloc(sizeof(*p));
    assert(p);
    p->a = 42;
    rcu_assign_pointer(gp, p);
}

void thread1(void) {
    struct rcutest *p;
    p = rcu_dereference(gp);
    gslp = p;
    p = gslp;
    if (p)
      assert(p->a == 42);
}

Figure 10: Litmus Test: Non-Local Storage For Dependency

that the assertion on line 19 never triggers, despite the store to and reload from gslp.
Note that this pattern must be used with care because dependency ordering is not guaranteed when the reload is done by some other thread, as shown by the next litmus test.

4.5 Non-Local Storage and Reload Kills Dependency

Figure 11 shows that storing to a non-local variable, then reloading from that variable within some other thread, kills the dependency. Dependency ordering therefore does nothing to prevent the assertion on line 25 from triggering.

That said, in many implementations this assertion will trigger with very low probability. One way to increase the probability of triggering is to arrange for the memory allocated by line 5 to be pre-loaded into thread2()’s cache.

4.6 Casting For Dependency

Figure 12 shows that casting a pointer to another pointer type with compatible layout preserves dependency ordering so that the assertion on line 19 never triggers, despite the cast operation.

4.7 Casting to Non-Pointer Kills Dependency

Figure 13 shows that casting a pointer to a non-pointer type can kill the dependency, despite the later cast back to a pointer type. The assertion on line 20 can therefore trigger.
1 void thread0(void)
2 {
3     struct rcutest *p;
4     p = (struct rcutest *)malloc(sizeof(*p));
5     assert(p);
6     p->a = 42;
7     rcu_assign_pointer(gp, p);
8 }
9
10 void thread1(void)
11 {
12     struct rcutest *p;
13     struct rcutest1 *q;
14     p = rcu_dereference(gp);
15     q = (struct rcutest1 *)p;
16     if (q)
17         assert(q->a == 42);
18 }
19
20 void thread2(void)
21 {
22     struct rcutest *p;
23     p = atomic_load_explicit(&gsgp, memory_order_relaxed);
24     if (p)
25         assert(p->a == 42);
26 }

Figure 11: Litmus Test: Non-Local Storage and Reload Kills Dependency

Figure 12: Litmus Test: Casting For Dependency
```c
void thread0(void)
{
    struct rcutest *p;
    p = (struct rcutest *)malloc(sizeof(*p));
    assert(p);
    p->a = 42;
    rcu_assign_pointer(gp, p);
}

void thread1(void)
{
    struct rcutest *p;
    uintptr_t q;
    p = rcu_dereference(gp);
    q = reinterpret_cast<uintptr_t>(p);
    p = (struct rcutest *)q;
    if (p)
        assert(p->a == 42);
}
```

Figure 13: Litmus Test: Casting to Non-Pointer Kills Dependency

```c
void thread0(void)
{
    struct rcutest *p;
    p = (struct rcutest *)malloc(sizeof(*p));
    assert(p);
    p->a = 42;
    rcu_assign_pointer(gp, p);
}

void thread1_help(struct rcutest *q)
{
    if (q)
        assert(q->a == 42);
}

void thread1(void)
{
    struct rcutest *p;
    p = rcu_dereference(gp);
    thread1_help(p);
}
```

Figure 14: Litmus Test: Function Argument Carries Dependency
void thread0(void) {
    struct rcutest *p;
    p = (struct rcutest *)malloc(sizeof(*p));
    assert(p);
    p->a = 42;
    rcu_assign_pointer(gp, p);
}

void thread1_help(struct rcutest *q [ [[carries_dependency]] ])
{
    if (q)
        assert(q->a == 42);
}

void thread1(void) {
    struct rcutest *p;
    p = rcu_dereference(gp);
    thread1_help(p);
}

Figure 15: Litmus Test: Function Argument Explicitly Carries Dependency

4.8 Function Argument Carries Dependency

Figure 14 shows that passing a pointer via a function argument preserves dependency ordering so that the assertion on line 14 never triggers. As shown in Figure 15, the [[carries_dependency]] attribute could be used to document the fact that the developer expects a dependency chain to pass into the function via this parameter, and implementations might use these attributes to improve diagnostics.

4.9 Function Return Carries Dependency

Figure 16 shows that returning a pointer from a function preserves dependency ordering so that the assertion on line 22 never triggers. As shown in Figure 17, the [[carries_dependency]] attribute can again be used to make the dependency-carrying explicit.

4.10 Array-Offset Dependency

Figure 18 shows how dependencies survive addition of integer offsets via array indexing. Dependency ordering guarantees that the assertion on line 18 never triggers.

4.11 Integer-Pointer Addition Dependency

Figure 19 shows how dependencies survive direct addition of integer offsets. Dependency ordering guarantees that the assertion on line 18 never triggers.
void thread0(void) {
  struct rcutest *p;
  p = (struct rcutest *)malloc(sizeof(*p));
  assert(p);
  p->a = 42;
  rcu_assign_pointer(gp, p);
}

struct rcutest *thread1_help(void) {
  return rcu_dereference(gp);
}

void thread1(void) {
  struct rcutest *p;
  p = thread1_help();
  if (p)
    assert(p->a == 42);
}

Figure 16: Litmus Test: Function Return Carries Dependency

void thread0(void) {
  struct rcutest *p;
  p = (struct rcutest *)malloc(sizeof(*p));
  assert(p);
  p->a = 42;
  rcu_assign_pointer(gp, p);
}

[[carries_dependency]] struct rcutest *thread1_help(void) {
  return rcu_dereference(gp);
}

void thread1(void) {
  struct rcutest *p;
  p = thread1_help();
  if (p)
    assert(p->a == 42);
}

Figure 17: Litmus Test: Function Return Explicitly Carries Dependency
1 void thread0(void)
2 {
3   int *p;
4   p = (int *)malloc(4 * sizeof(*p));
5   assert(p);
6   p[0] = 1;
7   p[1] = 42;
8   rcu_assign_pointer(gip, p);
9 }
10
11 void thread1(void)
12 {
13   int *p;
14   p = rcu_dereference(gip);
15   if (p)
16     assert(p[p[0]] == 42);
17 }
18
Figure 18: Litmus Test: Array-Offset Dependency

1 void thread0(void)
2 {
3   int *p;
4   p = (int *)malloc(4 * sizeof(*p));
5   assert(p);
6   p[0] = 1;
7   p[1] = 42;
8   rcu_assign_pointer(gip, p);
9 }
10
11 void thread1(void)
12 {
13   int *p;
14   p = rcu_dereference(gip);
15   if (p)
16     assert(*(p + p[0]) == 42);
17 }
18
Figure 19: Litmus Test: Integer-Pointer Addition Dependency
4.12 Integer-Pointer Subtraction Dependency

Figure 20 shows how dependencies survive direct subtraction of integer offsets. Dependency ordering guarantees that the assertion on line 18 never triggers.

4.13 Field-Selection Offset Dependency

Figure 21 shows that dependencies are carried through offsets due to the field-selection operator. Dependency ordering guarantees that the assertion on line 17 never triggers.
```c
void thread0(void) {
    int *p;
    p = (int *)malloc(sizeof(*p));
    assert(p);
    *p = 42;
    rcu_assign_pointer(gip, p);
}

void thread1(void) {
    int *p;
    p = rcu_dereference(gip);
    if (p)
        assert(*p == 42);
}
```

Figure 22: Litmus Test: Direct Dereferencing Dependency

4.14 Direct Dereferencing Dependency

Figure 22 shows how dependencies direct dereferencing. Dependency ordering guarantees that the assertion on line 17 never triggers.

4.15 Enclosing-Structure Location Dependency

Figure 23 shows that dependencies are carried through offsets used to locate an enclosing structure, an idiom used by the `container_of()` macro in the Linux kernel. Dependency ordering guarantees that the assertion on line 22 never triggers.

4.16 Conditional-Expression Dependency

Figure 24 shows a dependency carried through a conditional expression. Dependency ordering guarantees that the assertion on line 20 never triggers.

4.17 Comma-Expression Dependency

Figure 25 shows a dependency carried through the right-hand operand of a comma expression. Dependency ordering guarantees that the assertion on line 20 never triggers.

4.18 Equality Comparisons Kill Dependency

Figure 26 shows a dependency chain being killed by the equality comparison on line 19. The implementation is within its rights to substitute `i = rt.b` for line 20, which the implementation can hoist to precede line 19, even on a strongly ordered system. This hoisting allows access to pre-initialization values for `rt.b` which would be the constant 2 from the initializer on line 1. In this case, the assertion on line 23 would trigger.
1 void thread0(void)
2 {
3     struct rcutest1 *p;
4
5     p = (struct rcutest1 *)malloc(sizeof(*p));
6     assert(p);
7     p->a = 42;
8     rcu_assign_pointer(gp, &p->rt);
9 }
10
11 void thread1(void)
12 {
13     char *cp;
14     struct rcutest *p;
15     struct rcutest1 *q;
16
17     p = rcu_dereference(gp);
18     if (p) {
19         cp = (char *)p;
20         cp -= (uintptr_t)&((struct rcutest1 *)NULL)->rt;
21         q = (struct rcutest1 *)cp;
22         assert(q->a == 42);
23     }
24 }

Figure 23: Litmus Test: Enclosing-Structure Location Dependency

1 void thread0(void)
2 {
3     struct rcutest *p;
4
5     p = (struct rcutest *)malloc(sizeof(*p));
6     assert(p);
7     p->a = 42;
8     p->b = 43;
9     rcu_assign_pointer(gp, p);
10 }
11
12 void thread1(void)
13 {
14     int i;
15     struct rcutest *p;
16
17     p = rcu_dereference(gp);
18     if (p) {
19         i = (random() & 0x1) ? p->a : p->b;
20         assert(i == 42 || i == 43);
21     }
22 }

Figure 24: Litmus Test: Conditional-Expression Dependency
1 void thread0(void)  
2 {  
3     struct rcutest *p;  
4     p = (struct rcutest *)malloc(sizeof(*p));  
5     assert(p);  
6     p->a = 42;  
7     p->b = 43;  
8     rcu_assign_pointer(gp, p);  
9 }  
10 }  
11  
12 void thread1(void)  
13 {  
14     int i;  
15     struct rcutest *p;  
16     int junk;  
17     p = rcu_dereference(gp);  
18     if (p) {  
19         i = (junk = 1, p->a);  
20         assert(i == 42);  
21         assert(junk == 1);  
22     }  
23 }  
24 }

Figure 25: Litmus Test: Comma-Expression Dependency

1 struct rcutest rt = { 1, 2, 3 };  
2  
3 void thread0(void)  
4 {  
5     rt.a = -42;  
6     rt.b = -43;  
7     rt.c = -44;  
8     rcu_assign_pointer(gp, &rt);  
9 }  
10 }  
11 void thread1(void)  
12 {  
13     int i = -1;  
14     int j = -1;  
15     struct rcutest *p;  
16     p = rcu_dereference(gp);  
17     j = p->a;  
18     if (p == &rt)  
19         i = p->b; /* Dependency chain broken */  
20     else if (p)  
21         i = p->c;  
22     assert(i < 0);  
23     assert(j < 0);  
24 }  
25 }

Figure 26: Litmus Test: Equality Comparisons Kill Dependency
struct rcutest rt = { 1, 2, 3 };  
void thread0(void)  
{  
   rt.a = -42;  
   rt.b = -43;  
   rt.c = -44;  
   rcu_assign_pointer(gp, &rt);  
}  

void thread1(void)  
{  
   int i = -1;  
   int j = -1;  
   struct rcutest *p;  
   i = rt.b;  
   j = rt.a;  
   p = rcu_dereference(gp);  
   if (p && p != &rt) {  
      i = p->c;  
      j = p->a;  
   } else if (!p) {  
      i = j = -1;  
   }  
   assert(i < 0);  
   assert(j < 0);  
}

Figure 27: Litmus Test: Equality Comparisons Kill Dependency, Optimized

Worse yet, the implementation would be within its rights to transform the code as shown in Figure 27. In this case, both assertions can clearly trigger. In other words, an equality comparison can break dependencies in code preceding the dependency.

4.19 Equality Comparisons Without Killing Dependency

Figure 28 shows how pointer_cmp_eq_dep() and friends provide a way of carrying out comparisons without killing dependency chains. With the dependency chain preserved, the assertions on lines 23 and 24 cannot trigger.

5 Benefits, Drawbacks, Mitigations, and Open Issues

This section compares the proposal to the requirements and desiderata called out in Section 2.1.

As required, dependency chains do not depend solely on marking objects carrying dependencies. Such objects may be marked, if desired, using [[carries_dependency]], which can improve diagnostics and perhaps also improve formal verification.

The head of a dependency chain is marked with a memory_order_consume load. No other markings are needed, although [[carries_dependency]] may
Figure 28: Litmus Test: Equality Comparisons Without Killing Dependency

be used as noted above. In addition, `pointer_cmp_eq_dep()`, `pointer_cmp_ne_dep()`, `pointer_cmp_gt_dep()`, `pointer_cmp_ge_dep()`, `pointer_cmp_lt_dep()`, and `pointer_cmp_le_dep()`, if provided, could be used to avoid dependency-chain breakage that might otherwise occur due to value-narrowing comparisons.

Dependency chains have been defined to avoid the need for explicit memory-barrier instructions on mainstream systems.

Dependencies need be carried only through pointers and to non-pointers.

Implementations need not trace dependency chains. Again, `[[carries_dependency]]` may be used to permit improved diagnostics, but without the need to trace dependency chains.

It is important to note that this proposal does not address the need to tag pointers via low-order bits. Such tagging is required for a number of algorithms for concurrency and for memory allocation. Should a future proposal to define pointer-bit tagging be accepted, that proposal should accommodate carrying dependencies through tagged pointers.

In summary, this proposal provides the features needed by most existing RCU-related practice without placing undue burdens on the implementations.

However, there are some outstanding issues:

1. Formal verification very likely requires full annotation. However, requiring full annotation conflicts with both developer preferences and existing practice. This document therefore allows developers to omit annotations, but developers wishing to make full use of formal verification tools will need to add them.
2. The current definition of `memory_order_consume` is defined in terms of the
compiler’s ability to determine the exact value of a given pointer. It is not
yet clear how to word this, or, for that matter, whether any such definition
is appropriate for the standard.\(^7\)

3. It is generally agreed that some form of annotation should be made avail-
able, but the exact form is still subject to debate. This document uses the
`[[carries_dependency]]` annotation as a placeholder.

These issues will require some discussion.

References

[1] Arbel, M., and Morrison, A. Predicate rcu: An rcu for scalable con-
current updates. SIGPLAN Not. 50, 8 (Jan. 2015), 21–30.

mikeash.com: just this guy, you know?, May 2015.

[3] Compaq Computer Corporation. Shared memory, threads, interpro-


and Walpole, J. User-level implementations of read-copy update. IEEE
Transactions on Parallel and Distributed Systems 23 (2012), 375–382.

trees. ACM Transactions on Database Systems 5, 3 (September 1980), 354–
382.

using passive reader-writer locks. In 2014 USENIX Annual Technical
Conference (USENIX ATC 14) (Philadelphia, PA, June 2014), USENIX
Association, pp. 219–230.

Conference on Distributed Computing Systems (Washington, DC, USA,


\(^7\) That said, current implementations do work in exactly this manner.


Change Log

This paper was first posted informally in January of 2016. Revisions to this initial document are as follows:

- Add litmus tests in Section 4 as requested by Michael Wong. (January 8, 2016.)
- Apply feedback from Hans, including noting non-RCU use cases in the preamble starting on the first page, more clearly distinguishing this proposal from the wording in the current standard throughout, providing better justification for equality comparisons breaking dependency chains, adding Hans’s litmus test showing how avoiding undefined behavior can break dependency chains in a manner similar to equality comparisons, and marking as optional the wording for 29.9 (intrinsics for dependency-preserving equality comparison). (January 12, 2016.)
- Apply wording feedback from Jens Maurer. (January 13-22, 2016.)
- Apply feedback from Michael Wong, especially on litmus tests. (January 25, 2016 through February 5, 2016.)
- Highlight outstanding issues as discussed with Hans Boehm. (February 10, 2016.)
- Fix litmus tests issues reported by Michael Wong. (February 10, 2016.)

This version was mailed out as P0190R0 and presented at Jacksonville, Florida, USA. It was further revised:

- Fix litmus tests issues reported by Faisal Vali. (February 29, 2016.)
• Remove references to self-modifying code, as this is outside the standard, as noted by JF Bastien. (March 1, 2016.)

• Add note explicitly stating that overloaded operators are to be handled as function calls, as suggested by JF Bastien.

• Update wording based on detailed review with Jens Maurer and Jeffrey Yasskin. (March 4, 2016.)

This version was mailed out as P0190R1.

• Fix litmus-test issues reported by Michael Wong. (May 23, 2016).

• Update Michael Wong’s contact information. (May 25, 2016).

This version was mailed out as P0190R2.

• Update Michael Wong’s email address again. (May 27, 2016).

• Update to more accurately characterize DEC Alpha properties, per feedback from Linus Torvalds. (November 16, 2016).

• Add reference to P0462R1 for marked dependency chains.

This version was mailed out as P0190R3.

• Remove a stray backquote. (May 4, 2017.)

• Apply Will Deacon feedback. (May 4, 2017.)

• Globally indicate that A is of pointer type throughout wording. (July 14, 2017.)

This version was mailed out as P0190R4.